Dr. Vishwanath Karad,

MIT WORLD PEACE UNIVERSITY, PUNE

School of Computer Engineering and Technology

Lab Manual

SUBJECT NAME

Compiler Design Laboratory

Class: - TY BTech CSBS, Semester: - V

Prepared By: -
Prof. Shiv Sutar

Required H/W and S/W: - 64-bit Fedora or equivalent OS with 64-bit Intel-i5/ i7 or latest higher processor computers, FOSS tools, LEX, YACC, DAG, iburg, XMLVM.

Institute Vision:

Institute Mission:

Vision for Department of Computer Engineering:

Mission for Department of Computer Engineering:

CET2026B Compiler Design Laboratory

School of Computer Engineering and Technology	DEGREE:TY BTech CSBS: JULY 2023 – DEC 2023
COURSE: **Compiler Design Laboratory**	SEMESTER: V CC Assessment: 30 LC Assessment: 30
COURSE CODE: CET2026B	COURSE TYPE : **Core**/Elective
COURSE AREA/DOMAIN: ALGORITHM & DESIGN	CONTACT HOURS: 4 hours/Week
CORRESPONDING LAB COURSE CODE : CET2026B	LAB COURSE NAME: Compiler Design Laboratory

SYLLABUS

Assignment No.	Title of the assignment **Group A (**Mandatory Six Assignments**)**
1	Write a C program to tokenize given input C program and identify category of token (Content Beyond Syllabus)
2	Assignment to understand basic syntax of LEX Specifications, built-in functions and Variables. (Additional Assignment) i) Write a program to find out whether given input is a letter or digit ii) Write a program to find out whether given input is a noun, pronoun, verb, adverb, adjective or preposition iii) Write a program to read input from a file and find & replace a

	given string iv) Write a program to change case of given input (Upper, Lower, Sentence, Toggle) v) Write a program to read positive integers from a file and calculate average of it.
3	Implement Lexical analyzer for sample language using LEX. (Subset of C)
4	Assignment to study basic syntax of YACC, generate Arithmetic Calculator & Scientific Calculator using YACC. (Additional Assignment)
5	Parser for sample language using YACC (for loop / switch stmt / while loop / If loop)
6	Implement Intermediate Code generation for sample language using LEX and YACC. (If loop)
7	

Text Books

Sr. No.	Text Books
1	J. R. Levine, T. Mason, D. Brown, "Lex & Yacc", O'Reilly, 2000, ISBN 81-7366 -061-X.

COURSE OBJECTIVES:

COURSE OUTCOMES:

COURSE PRE-REQUISITES:

GAPS IN THE SYLLABUS - TO MEET INDUSTRY/PROFESSIONAL REQUIREMENTS:

TOPICS BEYOND CONTENT SYLLABUS/ADVANCED TOPICS/DESIGN:

SR. NO.	CONTENT DESCRIPTION	PO MAPPING
1	Write a C program to tokenize given input C program and identify category of token	
2	Assignment to understand basic syntax of LEX Specifications, built-in functions and Variables.	
3	Assignment to study basic syntax of YACC, generate Arithmetic Calculator & Scientific Calculator using YACC	

WEB SOURCE REFERENCES:

1. http://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc5
2. http://nptel.ac.in/courses/106104123/
3. http://nptel.ac.in/courses/106104072/
4. http://nptel.ac.in/courses/106108052/
5. http://nptel.ac.in/courses/106101060/

DELIVERY/INSTRUCTIONAL METHODOLOGIES:

√CHALK & TALK	√LCD/SMART BOARDS	√STUDENT ASSIGNMENT	STUD. SEMINARS
√WEB RESOURCES	ADD-ON COURSES		

ASSESSMENT METHODOLOGIES-DIRECT

ASSIGNMENTS	√STUD. LAB PRACTICES	STUD. SEMINARS	SIMPLE QUESTIONS
TESTS/MODEL EXAMS	MINI/MAJOR PROJECTS	√UNIV. EXAMINATION	CERTIFICATIONS
ADD-ON COURSES	IN TUTORIAL HOUR	OTHERS √ MOCK EXAM	

ASSESSMENT METHODOLOGIES-INDIRECT

√ASSESSMENT OF COURSE OUTCOMES (BY FEEDBACK, ONCE)	ASSESSMENT OF MINI/MAJOR PROJECTS BY EXT. EXPERTS	√STUDENT FEEDBACK ON FACULTY (TWICE)	OTHERS

Prepared By
Prof. Shiv Sutar

Approved By

Verified By

(Head of the School)

Experiment Number: 01 (CBS)

TITLE: Write a C program to tokenize given input C program and identify category of token.(Content beyond Syllabus).

PROBLEM STATEMENT:

Write a C program to tokenize given input C program and identify category of token

OBJECTIVES:

1. To understand the working of compiler (lexical analyzer) for C programs.

THEORY:

- To generate a lexical analyzer, two important things are needed. Firstly it will need a precise **specification of the tokens** of the language. Secondly it will need a specification of the **action to be performed on identifying each token**. For this operation several tools have been built which uses regular expression as an output, it generates a lexical analyzer. The particular tool of UNIX called **Lex** that has been widely used to specify lexical analyzer for a variety of languages.
- Lex is generally used as shown in Fig. 1.1.

First, a specification of a lexical analyzer is prepared. This specification used to write program which is having extension ***.l*** (e.g. **first.*l*, or ex.*l***). This program is written in lex language and run through the Lex compiler to produce C code in lex.yy.c. The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first *.l* or lex.*l*. (Finally lex.yy.c is run through the C compiler to produce object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens.).

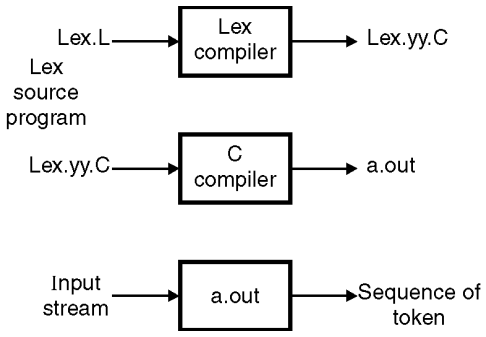


Fig. 1.1 How Lex Used & react for given program.

Lex Pattern :

Lex pattern are standard UNIX regular expressions using standard symbol which check the input stream (lexeme).

Standard regular expression	**Matches**	**Example**
c	Single character not operator	x
\c	Any character following the \ less its meaning and take literally	\ *
"S"	String S literally	"**"
.	Any single character except a newline (\n)	a . *h
^	Beginning of line	^abc
$	End of line	abc$
[S]	Any character in S	[abc], [A – Z]
[^S]	Any character except from S	[^abc]
r*	Zero or more occurrence	a*
r+	One or more occurrence	a+
r?	Zero or one r	0?
r {m, n}	m to n occurrence of r	a{1,5}
r1 r2	r1 then r2	ab
r1 | r2	r1 or r2	a : b
(r)	R	(a : b)
r1/r2	r1 when followed by r2	Abc/123

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Algorithm:-

1. Declaration of File pointer to read the input that contains sample C program.
2. Open the file in read/write mode.
3. Using string matching functions, identify various types of tokens like header file, identifier, keywords, in-built functions, operators etc.
4. Display the output.

Testing

Input : Already created any C File.

File Name : test.c

```
main()
        {
            int a,b; // variable declared
        }
```

Output :

```
(     : Delemeter
)     : Delemeter
int   : Keyword
a     : Identifier
b             : Identifier
;             : Delemeter
//varianble declared : Comments
```

Execute the program with the following commands.

```
$ cc assignment3.c
$ ./a.out test.c
```

FAQ's

1. What is the significance of lexical analyzer?
2. What is a token? What are the categories of token?
3. What are the different string functions used for pattern matching in an assignment?

Experiment Number: 01

TITLE: Assignment to understand basic syntax of LEX Specifications, built-in functions and Variables (Additional Assignment)

PROBLEM STATEMENT:

i) Write a program to find out whether given input is a letter or digit.
ii) Write a program to find out whether given input is a noun, pronoun, verb, adverb, adjective or preposition.
iii) Write a program to count number of lines, characters, words and vowels from given input.
iv) Write a program to read input from a file and find & replace a given string.
v) Write a program to change case of given input (Upper, Lower, Sentence, Toggle)

OBJECTIVES:

1. To understand first phase of compiler: Lexical Analysis
2. To learn and use compiler writing tools.

THEORY:

- To generate a lexical analyzer two important things are needed. Firstly it will need a precise **specification of the tokens** of the language. Secondly it will need a specification of the **action to be performed on identifying each token**. For this operation several tools have been built which uses regular expression as an output, it generates a lexical analyzer. The particular tool of UNIX called **Lex** that has been widely used to specify lexical analyzer for a variety of languages.
- Lex is generally used as shown in Fig. 1.1.

First, a specification of a lexical analyzer is prepared. This specification used to write program which is having extension ***.l*** (e.g. **first.*l*, or ex.*l***). This program is written in lex language and run through the Lex compiler to produce C code in lex.yy.c. The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first *.l* or lex.*l*. (Finally lex.yy.c is run through the C compiler to produce object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens.).

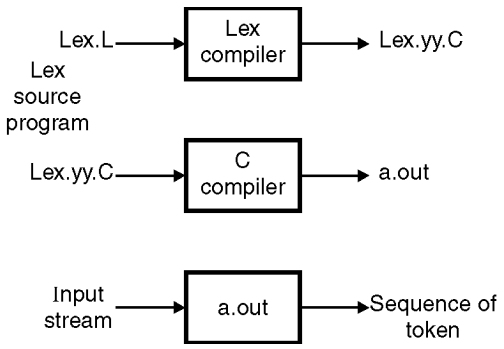


Fig. 1.1 How Lex Used & react for given program.

Lex Specifications :

- A lex program consists of three parts :

{declaration}

% %

{translation rules}

% %

{programmer subroutines}

- The first section of declaration includes declaration of variable, constants and regular definitions.
- Second section is for translation rules which consist of regular expression and action with respect to it. If regular definition is declared in declaration section, it uses that for regular expression. Regular declaration is declared as digit [0-9] in declaration section & used in { } brackets in translation rules. Example {digit} +.
- Every section end with % % symbol.
- The translation rules of a Lex program are statements of the form as follows:

r { action; }

- Here, each r is a regular expression and action is a program fragment describing what action to be taken when pattern matches.
- The actions are written in C language.
- The third section holds whatever program subroutines are needed by the actions. These procedures can be compiled separately and loaded later with the lexical

analyzer.

Lex Pattern :

Lex pattern are standard UNIX regular expressions using standard symbol which check the input stream (lexeme).

Standard regular expression	**Matches**	**Example**
c	Single character not operator	x
\c	Any character following the \ less its meaning and take literally	\ *
"S"	String S literally	"**"
.	Any single character except a newline (\n)	a . *h
^	Beginning of line	^abc
$	End of line	abc$
[S]	Any character in S	[abc], [A – Z]
[^S]	Any character except from S	[^abc]
r*	Zero or more occurrence	a*
r+	One or more occurrence	a+
r?	Zero or one r	0?
r {m, n}	m to n occurrence of r	a{1,5}
r1 r2	r1 then r2	ab
r1 | r2	r1 or r2	a : b
(r)	r	(a : b)
r1/r2	r1 when followed by r2	Abc/123

Lex Actions **:**

- Lex actions are C statements which are executed or actions are performed when regular expression pattern matches with lexeme. An action may be of single line C statement or multiple statements enclosed in {...} C brackets.
- For example :

```
%%
"india"     {
            printf ("India is great");
            }
%%
```

- Here India is a string, if matches with lexeme action take place and print “India is great”.
- Some important variables :
- **yylval** : Global variable which returns more information about lexeme to parser with the value in lexeme.
- **yytext** : The variable yytext carries point to the variable that we have been calling lexeme beginning that is a pointer to the first character of the lexeme.
- In declarations surrounded by % {and %} are declarations for manifest constants. A manifest constant is an identifier that is declared to represent a constant. Anything appearing between these brackets is copied directly into the lexical analyzer lex.yy.c and is not treated as part of the regular definitions or the translation rules.
- In third subroutine section we can write a user subroutines its option to user e.g. yylex( ) is a function automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Algorithm 1:- Write a program to find out whether given input is a letter or digit.

1. Declaration of File pointer to read the input. Declare the variable for letter and digit.
2. Declaration of header files.
3. Define the Regular Definition for letter and digit.
4. End of declaration section with %%
5. Write a regular expression for letter and digit and perform respective action in front of that in the second section of LEX.
6. End the translation rules section with %%.
7. In the last subroutines section write a main function which call the yylex. Open the file with declared file pointer to read. Assign the yyin pointer to the opened file.
8. Write a yywrap function to end the given program.

Testing:

Input : Any File with some text data or digit.

For Example:

File Name : input.txt

John123

Output:

J	: letter
o	: letter
h	: letter
n	: letter
1	: digit
2	: digit
3	: digit

OR

John	: letter
123	: digit

Algorithm 2:- Write a program to find out whether given input is a noun, pronoun, verb, adverb, adjective or preposition

1. Declaration of File pointer to read the input. Declare the variable for noun, pronoun, verb, adverb, adjective or preposition.
2. Declaration of header files.
3. Define the Regular Definition for noun, pronoun, verb, adverb, adjective or preposition.
4. End of declaration section with %%
5. Write a regular expression for noun, pronoun, verb, adverb, adjective or preposition and perform respective action in front of that in the second section of LEX.
6. End the translation rules section with %%.
7. In the last subroutines section write a main function which call the yylex. Open the file with declared file pointer to read. Assign the yyin pointer to the opened file.
8. Write a yywrap function to end the given program.

Testing:

Input : Any File with some text data.

For Example:

File Name : input.txt

She is good girl.

Output:

She	: Pronoun
is	: verb
good	: adjective
girl	: noun

Algorithm 3:- Write a program to count number of lines, characters, words and vowels from given input.

1. Declaration of File pointer to read the input. Declare the variable for character, line , word, vowels Counter.
2. Declaration of header files.
3. Define the Regular Definition for Word, White Space.
4. End of declaration section with %%
5. Write a regular expression for white space, vowels, line no, character, word count & perform respective action in front of that in the second section of LEX.
6. End the translation rules section with %%.
7. In the last subroutines section write a main function which call the yylex. Open the file with declared file pointer to read. Assign the yyin pointer to the opened file.
8. Write a yywrap function to end the given program.

Testing:

Input : Any File with some text data.

For Example:

File Name : input.txt

```
main()
{
  printf(" Hello to all");
}
```

Output:

No. of Lines	: 04
No. of Characters	: 30
No. of Words	: 05
No. of Vowels	: 07

After Remove the White Space from the program is:

```
main()
{
  printf("Helltoall");
}
```

Algorithm 4:- Write a program to read input from a file and find & replace a given string.

1. Declaration of File pointer to read the input & other File pointer to write the output. Declare the variable for read string for find & other for replace the string.
2. Declaration of header files.
3. Define the Regular Definition for alphabets.
4. End of declaration section with %%
5. Write a regular expression who read the string.
6. The action in front of this regular expression is if the string is found in the file then replaces it & store in output file. Otherwise rest of the contains are copy as it is in output file.
7. End the translation rules section with %%.
8. In the last subroutines section write a main function which call the yylex. Open the file with declared file pointer to read. Assign the yyin pointer to the opened file.
9. Write a yywrap function to end the given program.

Testing:

Input: Any File with some text data.
For Example:
File Name : input.txt

```
main()
{
  printf(" Hello to all");
}
```

Output:

```
Enter the string to find      : Hello
Enter the string to replace: Bye
main()
{
  printf(" Bye to all");
}
```

Algorithm 5:- Write a program to change case of given input (Upper, Lower, Sentence, Toggle)

1. Declaration of variable to read the input choice to change the case & other variable for read the string which case can be change.
2. Declaration of header files.
3. End of declaration section with %%
4. Write a regular expression who read the choice.
5. The action in front of this regular expression
    1. To change case to Upper case check that string character are in between 97 to 122. If it is subtract by 32 value from yytext.
    2. To change case to lower case check that string characters are in between 65 to 97 then add by 32 to yytext.
    3. If case is Sentence case then only First character of yytext is going to convert to Upper case by checking it's ASCII value & as per range – by 32 / keep as it. Rest of the string is going to convert to lower case.
    4. For Toggle case characters in upper convert to lower & vice versa as per it's ASCII value.
6. End the translation rules section with %%.
7. In the last subroutines section write a main function which call the yylex. Read the string to change the case.
8. Write a yywrap function to end the given program.

Testing

Input :

1. Upper Case
2. Lower Case
3. Toggle Case
4. Sentence Case

Enter the choice: 1
Enter the String : Hello

Output :

The string is : HELLO

Input :

1. Upper Case
2. Lower Case
3. Toggle Case
4. Sentence Case

Enter the choice: 2
Enter the String : HELLO

Output :

The string is : hello

Input :

1. Upper Case
2. Lower Case
3. Toggle Case
4. Sentence Case

Enter the choice: 3
Enter the String : Hello

Output :

The string is : hELLO

Input :

1. Upper Case
2. Lower Case
3. Toggle Case
4. Sentence Case

Enter the choice: 4
Enter the String : HELLO

Output :

The string is : Hello

Execute the program with the following commands.

```
$ lex first.l
$cc lex.yy.c –ll
$./a .out
```

FAQ's

1. What are compiler phases?
2. What are the different tasks of Lexical Analysis?
3. Explain regular expression of lex file.

Experiment Number: 02

TITLE: Implement Lexical analyzer for sample language using LEX. (Subset of C)

PROBLEM STATEMENT:

Implement a lexical analyzer for a subset of C using LEX. Implementation should support Error handling.

OBJECTIVES:

1. To understand the working of compiler for C programs.
2. To understand best practices in coding.

THEORY:

- To generate a lexical analyzer two important things are needed. Firstly it will need a precise **specification of the tokens** of the language. Secondly it will need a specification of the **action to be performed on identifying each token**. For this operation several tools have been built which uses regular expression as an output, it generates a lexical analyzer. The particular tool of UNIX called **Lex** that has been widely used to specify lexical analyzer for a variety of languages.
- Lex is generally used as shown in Fig. 1.1.

First, a specification of a lexical analyzer is prepared. This specification used to write program which is having extension ***.l*** (e.g. **first.*l*, or ex.*l***). This program is written in lex language and run through the Lex compiler to produce C code in lex.yy.c. The program lex.yy.c basically consists of a transition diagram constructed from the regular expressions of first *.l* or lex.*l*. (Finally lex.yy.c is run through the C compiler to produce object program a.out, and lexical analyzer transforms an input streams into a sequence of tokens.).

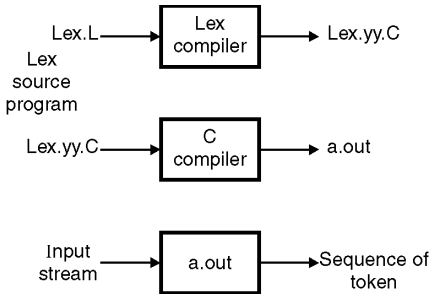


Fig. 1.1 How Lex Used & react for given program.

Lex Specifications :

- A lex program consists of three parts :

{declaration}

% %

{translation rules}

% %

{programmer subroutines}

- The first section of declaration includes declaration of variable, constants and regular definitions.
- Second section is for translation rules which consist of regular expression and action with respect to it. If regular definition is declared in declaration section, it uses that for regular expression. Regular declaration is declared as digit [0-9] in declaration section & used in { } brackets in translation rules. Example {digit} +.
- Every section end with % % symbol.
- The translation rules of a Lex program are statements of the form as follows:

r { action; }

- Here, each r is a regular expression and action is a program fragment describing what action to be taken when pattern matches.
- The actions are written in C language.
- The third section holds whatever program subroutines are needed by the actions. These procedures can be compiled separately and loaded later with the lexical analyzer.

Lex Pattern :

Lex pattern are standard UNIX regular expressions using standard symbol which check the input stream (lexeme).

Standard regular expression	**Matches**	**Example**
c	Single character not operator	x
\c	Any character following the \ less its meaning and take literally	\ *
"S"	String S literally	"**"

.	Any single character except a newline (\n)	a . *h
^	Beginning of line	^abc
$	End of line	abc$
[S]	Any character in S	[abc], [A – Z]
[^S]	Any character except from S	[^abc]
r*	Zero or more occurrence	a*
r+	One or more occurrence	a+
r?	Zero or one r	0?
r {m, n}	m to n occurrence of r	a{1,5}
r1 r2	r1 then r2	ab
r1 | r2	r1 or r2	a : b
(r)	R	(a : b)
r1/r2	r1 when followed by r2	Abc/123

Lex Actions **:**

- Lex actions are C statements which are executed or actions are performed when regular expression pattern matches with lexeme. An action may be of single line C statement or multiple statements enclosed in {...} C brackets.
- For example :

```
%%
 "india"      {
              printf ("India is great");
              }
 %%
```

- Here India is a string, if matches with lexeme action take place and print "India is great".
- Some important variables :
- **yylval** : Global variable which returns more information about lexeme to parser with the value in lexeme.
- **yytext** : The variable yytext carries point to the variable that we have been calling lexeme beginning that is a pointer to the first character of the lexeme.
- In declarations surrounded by % {and %} are declarations for manifest constants. A manifest constant is an identifier that is declared to represent a constant. Anything appearing between these brackets is copied directly into the lexical analyzer lex.yy.c and is not treated as part of the regular definitions or the translation rules.

- In third subroutine section we can write a user subroutines its option to user e.g. yylex( ) is a function automatically get called by compiler at compilation and execution of lex program or we can call that function from the subroutine section.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Algorithm:-

1. Declaration of File pointer to read the input. Declare the variable for line counter & choice to continue(as getch() function is not available in Linux we used scanf of choice variable to break the output till user enter single character).
2. Declaration of header files.
3. Define the Regular Definition for Delemeter & Integer.
4. End of declaration section with %%
5. Write a regular expressions who read various syntax of C subset like
   a. ("//"[^\n\r]*)(\n)) for Comments
   b. [+-]?{INT}\.{INT}([eE][+-]?{INT})? For Floating point number
   c. [+-]?{INT} for Integer Number
   d. (\"(^\n]*)\") For String etc.
6. The action in front of this regular expression is print which kind of token found then check for number of lines already displayed on the monitor (maximum is 40 lines) & used scanf to hold the screen till user enter the character.
7. End the translation rules section with %%.
8. In the last subroutines section write a main function with command line arguments which call the yylex.
9. Open the file available in argv variable with declared file pointer to read the input & Assign with yyin pointer.
10. Write a yywrap function to end the given program

Testing

Input : Already created any C File.

File Name : test.c

```
main()
        {
            int a,b; // variable declared
        }
```

Output :

```
(    : Delemeter
)    : Delemeter
int  : Keyword
a    : Identifier
```

```
  b            : Identifier
  ;            : Delemeter
//varianble declared : Comments
```

Execute the program with the following commands.

```
$ lex second.l
$cc lex.yy.c –ll
$ ./a.out test.c
```

FAQ's

1. How to declare regular definition in lex file?
2. How yylex() routine is called?
3. What is the significance of yywrap() routine?

Experiment Number: 03

TITLE: Assignment to study basic syntax of YACC, generate Arithmetic Calculator & Scientific Calculator using YACC. (Additional Assignment)

PROBLEM STATEMENT:
Generate Arithmetic Calculator & Scientific Calculator using YACC.

OBJETIVES:

1. To understand second phase of compiler: Syntax Analysis
2. To understand working of YACC utility

THEORY:

Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for "**Y**et **A**nother **C**ompiler **C**omplier".

YACC specification:

1. File in which parser generated is with **.**y extension.
2. e.g. parser ·y, which is containing YACC specification of the translator. After complete specification UNIX command.

   YACC parser ·y

   transforms the file parser ·y into a C program called y**.**tab·C using LR parser. The program y·tab·**c** is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. By compiling y·tab·**c** along with the by library that contains the LR parsing program using the command.

   CC y·tab·c – ly
3. We obtain the desired object program a·out as shown in following figure
4. A YACC source program has three parts.

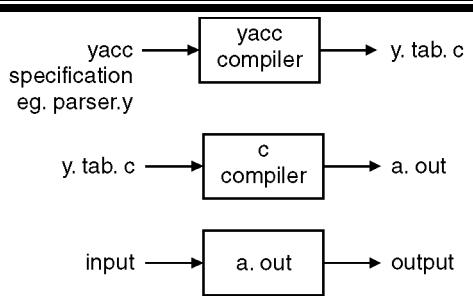
   Declaration

   %%

   Translation rules

   %%

   supporting subroutines



Automatic generation of the LR parser :

Definition section (Declaration) :

The definitions and programs section are optional. Definition section handles control information for the YACC-generated parser and generally set up the execution environment in which the parser will operate.

Declaration part :

In declaration section, % { and %} symbol used for C declaration. This section is used for definition of token, associativity and precedence of operator. The statement between % { and % } is passed as it is to C program, normally used for comments.

For token declaration the statement is:

% token DIGIT

which declares DIGIT to be token.

Associativity and precedence defines like

% left '*' '/'

which declare operator are left associative and * and / are having same precedence than +.

Tokens declared in this section can be used in the second and third part of YACC specification.

% token SUM

% token LOG

YACC program is produced by command $ YACC demo.y -d (Here program name is supposed as demo with extension ·y). Output of this command generate a file y·tab·h which contains information about token declaration of first section.

e.g.

define SUM 100

define LOG 112

Translation :

Rule section :

In YACC specification after the first %% pair, we put the translation rules. Each rule consists of a grammar production and the associated semantic action. It means that YACC rules define what is a legal sequence of tokens in our specifications language.

A set of productions

<left side> → <alt 1> | <alt 2> | … <alt n>

can be written in YACC as

```
<left side>     :    <alt 1>    {action 1}
                |    <alt 2>    {action 2}
                …
                |    <alt n>    {action n}
                ;
```

```
e.g. menu item    :    LABEL EX
                  ;
```

- This rule defines a non-terminal symbol, menu item in terms of the two tokens LABEL and EX. Tokens are also called as "terminal symbol" because parser does not need to expand them further.
- Conversely menu item is a "non-terminal symbol" because it can be expanded into LABEL and EX. These two parts of semantic rule is separated by vertical bar and a semicolon follows each left side with its alternatives and their semantic action.
- The first left side is token to be the starting symbol.
- A YACC semantic action is a sequence of statements. In a semantic action, the symbol $$ refers to the attribute value associated with the non-terminal of the left while $i refer to the value associated with the ith grammar symbol.

  e.g. The two E-productions

  E → E + T/T in YACC

```
exp     :    exp '+' term {$$ = $1 + $3;}
        |    term
        ;
```

- In above production exp is $1, '+' is $2 and term is $3. The semantic action associated with first production adds values of exp and term and result of addition copying in $$ (exp) left hand side. For above second number

production, we have omitted the semantic action since it is just copying the value.

- {$$ = $1;} is the default semantic action.

❖ **Token types :**

- Token data types are declared in YACC using the YACC declaration % union, like this :

```
% union {
        char * str ;
        int num ;
      }
```

- This variable data type is required when token is holding some value and we have to specify which kind of value it is holding.
- Normally yylval is being defined a union as types (char*) and int.
- We use this variable declaration to specify the type which is associated with token in following manner :

```
% token <str> EXE
% token <num> DIGIT
```

- Note that we have the token value, we want to use it. This value is used by YACC in above maintained variables.

  e.g. $$, $1 …etc.

- This token declaration is in YACC declaration section.

```
% type <num> default
```

- This is same approach as we used for % token definitions but this is not used by the lex.

❖ **Subroutines :**

- YACC generates a single function called yyparse ( ). This function requires no parameters and returns either a 0 on success, and 1 on failure. If syntax error over its return 1.
- The special function yyerror ( ) is called when YACC encounters an invalid syntax. The yyerror ( ) is passed a single string (char*) argument. This function just prints "parse error" message, it is possible to give your own message in this function like

```
yyerror (char* err)
{
    fprintf (stderr, "% S \ n", error);
}
```

```
yyerror (char* err)
{
  printf ("Divide by zero");
}
```

- When lex and YACC work together lexical analyzer yylex ( ) to produce pairs consisting of a token and its associated attribute value.
- If a token such as DIGIT is returned, the token value associated with a token is communicated to the parser through a YACC defined variable yylval.
- We have to return tokens from lex to YACC, where its declaration is in YACC. To link this lex program include a y · tab · h file, which is generated after YACC the program. Consider this given program of calculator.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Arithmetic Calculator:

Algorithm:-

LEX

1. Declaration of header files specially y.tab.h which contain declaration for num.
2. End declaration section by %%
3. Match regular expression for float number to input number
4. If match found then convert it into float and store it in yylval.p where p is pointer declared in YACC
5. Return token num which indicate nonterminal
6. If input contains tab (\t) then do nothing
7. If input contains new line character (\n) then return 0
8. If input contains '.' then return yytext[0]
9. End rule-action section by %%
10. Declare main function
    a. open file given at command line
    b. if any error occurs then print error and exit
    c. assign file pointer fp to yyin
    d. call function yylex until file ends
11. End

YACC

1. Declaration of header files
2. Declare pointer of type double in union
3. Declare token num of type pointer p
4. Declare type exp of type p
5. Give precedence to '*','/'
6. Give precedence to '+','-'
7. End of declaration section by %%
8. If final expression evaluates then print the answer
9. If input type is expression of the form
    a. exp'+'exp then add the two numbers
    b. exp'-'exp then subtract the two numbers
    c. '('exp')' then assign value to exp
    d. Num then do nothing
10. End the section by %%
11. Declare file *yyin externally
12. Declare main function
    a. call yyparse function untill yyin ends
13. Declare yyerror for if any error occurs
14. Declare char pointer s to print error
15. Print error message.
16. End of the program.

Testing

Input : 3+2*5

Output : 25

Input : sqrt(9)

Output : 3

Scientific Calculator:

Algorithm:-

LEX

1. Declaration of header files specially y.tab.h which contain declaration for num, SIN, COS.
2. End declaration section by %%
3. Match regular expression for float number to input number
4. If match found then convert it into float and store it in yylval.p where p is pointer declared in YACC
5. Return token num which indicate nonterminal
6. If input contains sine operation then return token SIN to YACC
7. If input contains cosine operation then return token COS to YACC
8. If input contains tab (\t) then do nothing
9. If input contains new line character (\n) then return 0
10. If input contains '.' then return yytext[0]
11. End rule-action section by %%
12. Declare main function
    a. Open file given at command line
    b. If any error occurs then print error and exit
    c. Assign file pointer fp to yyin
    d. Call function yylex until file ends
13. End

YACC

1. Declaration of header files
2. Declare pointer of type double in union
3. declare token num of type pointer p
4. Declare tokens SIN, COS
5. Declare type exp of type p
6. Give precedence to '*','/'
7. Give precedence to '+','-'

8. End of declaration section by %%
9. If final expression evaluates then print the answer
10. If input type is expression of the form
    a. exp'+'exp then add the two numbers
    b. exp'-'exp then subtract the two numbers
    c. '('exp')' then assign value to exp
    d. SIN then calculate sine of the value and assign to exp
    e. COS then calculate cosine of the value and assign to exp
    f. Num then do nothing
11. End the section by %%
12. Declare file *yyin externally
13. Declare main function
    a. call yyparse function untill yyin ends
14. Declare yyerror for if any error occurs
15. Declare char pointer s to print error
16. Print error message.
17. End of the program.

Testing :-

Input : sin(90)

Output : 1

Input : cos(1)

Output : 0.99

Execute the program with the following commands.

```
$ yacc –d parse.y
$ lex parse.l
$cc lex.yy.c y.tab.c –ll –ly -lm
$ ./a.out
```

FAQ's

1. What is the role of parser? YACC is which kind of a parser?
2. How the tokens generated from lex are passed to YACC?
3. Why y.tab.h file is included in lex file?

Experiment Number: 04

TITLE: Implement Parser for sample language using YACC.

PROBLEM STATEMENT:

Implement YACC for Subset of C (for loop / switch stmt/ while loop/ If loop)

OBJECTIVES:

1.) To understand the basic syntax of YACC

2.) To understand how a compiler parsing phase works for subset of C.

THEORY:

Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for "**Y**et **A**nother **C**ompiler **C**omplier".

A YACC source program has three parts.

Declaration

%%

Translation rules

%%

supporting subroutines

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Algorithm:-

LEX

1. Declaration of header files specially y.tab.h which contains declaration for the tokens **FOR , OPBR ,CLBR ,SEMIC, RELOP, EQU, ID, NUM RELOP, INC, DEC**
2. Define the Regular Expression for the tokens **FOR , OPBR ,CLBR ,SEMIC, RELOP, EQU, ID, NUM RELOP, INC, DEC**
3. End of declaration section with %%
4. Match regular expression.

5. If match found then store it in yylval where p is pointer declared in YACC
6. Return the token.
7. End rule-action section by %%
8. End

```
//********************lex file*********************//

%{
     #include "y.tab.h"
     extern int yylval;
%}

%%
for {return (FOR);}
"(" {return (OPBR);}
")" {return (CLBR);}
";" {return (SEMIC);}
"=" {return (EQU);}
"<"|">" {return (RELOP);}
"++" {return (INC);}
"--" {return (DEC);}
[a-zA-Z]+ {yylval=yytext[0];return(ID);}
[0-9]+ {yylval=atoi(yytext);return(NUM);}
%%

yywrap()
{
  return 1;
}
```

YACC

1. Declaration of header files
2. Declare tokens **FOR , OPBR ,CLBR ,SEMIC, RELOP, EQU, ID,**
    a. **NUM RELOP, INC, DEC**
3. End of declaration section by %%
4. Declaration of grammar
5. End the section by %%
6. Declare main() function
    a. Call yyparse() function until yyin ends.
7. End of the program.

```
//************yacc file***************//

%{
     /*Program for YACC Specification*/
     #include <stdio.h>
     int flag=0;
%}
```

```
%token FOR OPBR CLBR SEMIC RELOP EQU ID NUM RELOP INC DEC
%%
S:FOR OPBR E1 SEMIC E2 SEMIC E3 CLBR
{printf("Accepted!");flag=1;}
 ;

E1: ID EQU ID
  | ID EQU NUM
  ;
E2: ID RELOP ID
  | ID RELOP NUM
  ;
E3: ID INC
  | ID DEC
  ;

%%
main()
{
     yyparse();
}

yyerror(const char *msg)
{
     if(flag==0);
     printf("Not Accepted!");
}
```

TESTING:

```
INPUT
for(i=0;i<9;i++)
OUTPUT :Accepted!

INPUT:
for(i=0i<9;i++):
OUTPUT : Not Accepted!
```

FAQ's

1. When does yywrap function return 1 and what is data type of yyin?
2. What is the significance of YACC?
3. For which phase of compilation is YACC used ?

Experiment Number: 05

TITLE: Implement a parser for an expression grammar using LEX and YACC.

Experiment Number: 06

TITLE: Implement Intermediate Code generation for sample language using LEX and YACC.

PROBLEM STATEMENT:

Write a Program for Intermediate Code Generation for subset of C (If loop) using LEX & YACC

OBJECTIVES:

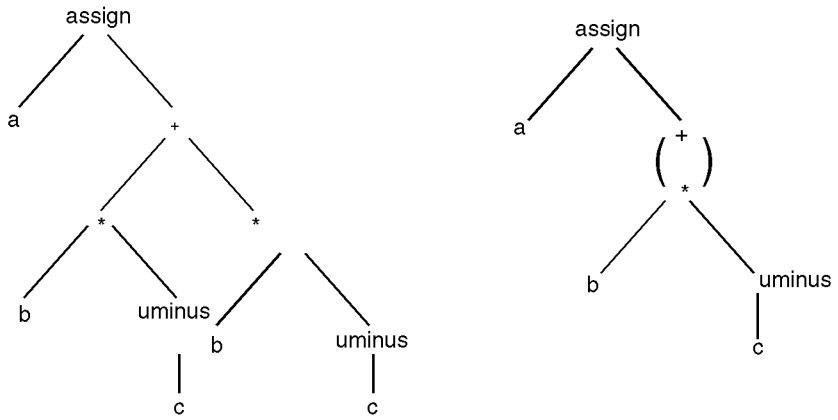
1) To understand the different forms of ICG
2) To understand how a compiler ICG phase works for subset of C.

THEORY:

- Syntax trees and postfix notation are two kinds of intermediate representations.
- A third called three address code will also be used.
- The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees for generating postfix notation.

6.2.1 Graphical Representations :

- A syntax tree depicts the natural hierarchical structure of a source program.
- A dag gives the same information but in compact way because common subexpressions are identified.
- A syntax tree and dag for the assignment statement a : = b * – c + b * – c appear in Fig. 6.2.1.



(a) Syntax tree **(b) DAG**

Fig. 6.2.1 : Graphical representation at a : = b * – c + b * – c

- Postfix notation is a linearized representation of a syntax tree, it is a list of the

  nodes of a tree in which a node appears immediately after its children.
- The postfix notation for the syntax tree in Fig. 6.2.1(a) is

  a bc uminus * bc uminus the *+ assign

Three · Address Code :

- Three address code is a sequence of statement of the general form

  x : y op z

  where x, y and z are names constants or compiler generated temporaries.

  op stands for any operator, such as fixed-or-floating point arithmetic operator

or logical operator on Boolean valued data.

- No built up arithmetic expressions are permitted, as there is only one operator

  on the right side of a statement.
- A source language expression like x + y * z might be translated into a sequence. t1 : y * z

  t2 : x + t1

  where t1, t2 are compiler-generated temporary names.
- Three –address code is a linearized representation of a syntax tree or a dag

  in which explicit names correspond to the interior nodes of the graph.
- The syntax tree and dag in Fig. 6.2.1 are represented by the three-address code

  sequence in Fig. 6.2.4.Variables names can appear directly in three-addres

  statements so Fig. 6.2.4(a) has no statements corresponding to the leaves in Fig.

Implementations of Three-Address Statements :

- A three-address is an abstract form of intermediate form and these statements

  can be implemented as records with fields for the operator and the operands.
- There are three representations

  a) Quadraple

b) Triples

c) Indirect triples

a) Quadraples :

- A quadraple is a record structure with four fields, which we will call as op arg1, arg2 and result.
- The op field contains an internal code for the operator.
- The three-address statement x : = y op 2 is represented by placing y in arg1, z in arg2, and x in result.
- Statements with unary operators like x : = – y or x : = y do not use arg 2.
- Operator like param use neigher arg2 not result.
- Conditional and unconditional jumps put the target label in result.
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. So, temporary names must be entered into the symbol table as they are created.
- The quadraples for the assignment a : b * – c + b * – c are shown in Fig..

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	B	t3	t4
(4)	+	t2	t4	t5
(5)	: =	t5		a

Fig.: Quadraples representation of three-address statement

b) Triples :

- Triples is a record structure with three field arg1, arg2 and op.
- The fields arg1 and arg2 for the arguments of op are either pointers to the symbol table or pointers into the triple structure.
- Triples corresponds to the representation of a syntax tree or dag by an array of nodes.
- Parenthesized numbers represent pointers into the triple structure. While

symbol-table pointers are represented by the names themselves.

- Fig. shows the triples for the assignment statement a : = b * – c + b * – c.

	op	Arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Fig.: Triple representation of three-address statement

- Fig. shows more representations for the ternary operation like x [i] : = y, it requires two entries in the triple structure, while x : = y [i] is naturally represented as two operations.

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

(a) x [i] : = y

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : y [i]

Fig. 6.2.9 : More triple representations

c) Indirect triples :

Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves.

- Let us use an array statement to list pointers to triples in the desired order.
- Then the triples in Fig. 6.2.8 be represented as in Fig. 6.2.10.

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	**arg1**	**arg2**
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Fig.: Indirect triples representation of three address statements

IMPLEMENTATION DETAILS / DESIGN LOGIC:

LEX

1. Declaration of header files specially y.tab.h which contains declaration for the tokens **RELOP ID IF BLST BLEND NUM A**
2. Define the Regular Expression for the tokens **RELOP ID IF BLST BLEND NUM A**.
3. End of declaration section with %%
4. Match regular expression.
5. If match found then store it in yylval
6. Return the token.
7. End rule-action section by %%
8. End

```
//********************lex specification**************//



%{
#include<stdio.h>
#include<string.h>
#include"y.tab.h"
#include<math.h>
%}

%%
[/t]+ ;
if { strcpy(yylval.cval,yytext);return IF;}
[{] { strcpy(yylval.cval,yytext);return BLST;}
[}] { strcpy(yylval.cval,yytext);return BLEND;}
[;] { strcpy(yylval.cval,yytext);return SEMI;}
[0-9]+ {yylval.dval=atoi(yytext); return NUM;}
[a-z]+ {strcpy(yylval.cval,yytext);return ID;}
"+"|"-"|"="|"*"|"/" {return *yytext;}
"("|")" {return *yytext;}
"<"|">" {strcpy(yylval.cval,yytext);return RELOP;}
\n {return *yytext;}
%%

yywrap()
{
```

```
return 1;
}
```

YACC

1. Declaration of header files
2. Declare structure for three address code with fields for operator 1 , operator 2
   operand , temporary variable.
3. Declare tokens **RELOP ID IF BLST BLEND NUM A**
4. End of declaration section
5. Declaration of grammar
6. End the section by %%
7. Declare main() function
   a. Call yyparse() function .
8. End of the program.

```
//**********************yaccspecification***************//

%{
      #include<stdio.h>
      #include<math.h>
      #include<string.h>

struct quad
{

char op[10];
char arg1[10];
char arg2[10];
char res[10];
} quad[20];

int blk_cnt=1,cnt=1;
char arr[10],str[10],temp[10];
struct block
{
  int st,end,if_flag;
} blk[20];

%}

%union
{
    char cval[10];
    int dval;
}
%token <cval> RELOP ID IF BLST BLEND
```

```
%token <dval> NUM
%type <cval> A
%left '-' '+'
%left '*' '/'
%left RELOP


%%
S : B '(' E ')' S1 '\n'  {print();}
  ;
B : IF {blk[blk_cnt].if_flag=1;}
  ;
E : ID RELOP ID {strcpy(quad[cnt].op,$2);
                 strcpy(quad[cnt].arg1,$1);
                 strcpy(quad[cnt].arg2,$3);
                 arr[0]=arr[0]+1;
                 arr[1]='\0';
                 strcpy(str,"t");
                 strcpy(quad[cnt].res,str);
                 cnt++;
                }
  ;

S1 : BLST1 AS BLEND1
   ;
AS : AS1
   ;
AS1 : ID '=' A {  strcpy(quad[cnt].op,"=");
                 strcpy(quad[cnt].arg1,$3);
                 strcpy(quad[cnt].arg2,"-");
                 strcpy(quad[cnt].res,$1);
                 cnt++;
                }
    ;

A : ID {strcpy($$,$1);}
  | NUM {sprintf(temp,"%d",$1); strcpy($$,temp);}
  | A '+' A { strcpy(quad[cnt].op,"+");
                 strcpy(quad[cnt].arg1,$1);
                 strcpy(quad[cnt].arg2,$3);
                 arr[0]=arr[0]+1;
                 arr[1]='\0';
                 strcpy(str,"t");
                 strcat(str,arr);
                 strcpy(quad[cnt].res,str);
                 cnt++;
                 strcpy($$,str);
                }
  ;
```

```
BLST1 : BLST { if(blk[blk_cnt].if_flag==1)
                {
                     blk[blk_cnt].st=cnt;
                     strcpy(quad[cnt].op,"if");
                     strcpy(str,"t");
                  strcat(str,arr);
                  strcpy(quad[cnt].arg1,str);
                  temp[0]=cnt+'2';
                  temp[1]='\0';
                   strcpy(quad[cnt].res,temp);
                  cnt++;
                   strcpy(quad[cnt].op,"goto");
                  cnt++;
                  }
               }
        ;
BLEND1 : BLEND { if(blk[blk_cnt].if_flag==1)
                   {
                      blk[blk_cnt].if_flag=0;
                      blk[blk_cnt].end=cnt;
                      temp[0]=cnt+'0';
                      temp[1]='\0';

strcpy(quad[blk[blk_cnt].st+1].res,temp);
                    }
                 blk_cnt++;
              }
         ;


%%
main()
{
yyparse();
}
yyerror()
{
 printf("ERROR");
 return 1;
}

print()
{
 int i;
 printf("\nS.No\tOpr\tArg1\tArg2\tResult\n");
 printf("================================================”);
 for(i=1;i<cnt;i++)
 {
```

```
   printf("\n%d\t%s\t%s\t%s\t%s",i,quad[i].op,quad[i].arg1,
quad[i].arg2,quad[i].res);
 }
 printf("\n %d",i);
}
```

TESTING

INPUT :

```
if(a<b) {a=b+1;};
```

OUTPUT

S.No	Opr	Arg1	Arg2	Result
1	<	a	b	t
2	if	ta	-	4
3	goto	-	-	6
4	+	b	1	tb
5	=	tb	-	a
6				

FAQ's

1.) What is the role/ significance of ICG in compilation?
2.) What are the different linear and graphical representations of Intermediate Code?
3.) What are different representations of 3 address code?
4.) What is the difference between syntax tree and DAG?

Experiment Number: 07

TITLE: Code optimization using DAG

PROBLEM STATEMENT:

Implement code optimization using DAG

OBJECTIVES:

1. To understand the working of code Optimization for Code Improvement.
2. To understand best practices and Techniques applied for Optimization.
3. To understand the working of Common Sub-Expression Elimination and Constant Expression Evaluation.

THEORY:

- Code Optimization is the phase of compilation that focuses on generating a good code.
- Most of the time a good code means a code that runs fast. However, there are some cases where a good code is a code that does not require a lot of memory.

Local Optimization

- Local Optimization focuses on:
    - Elimination of redundant operations
    - Effective instruction scheduling
    - Effective register allocation.

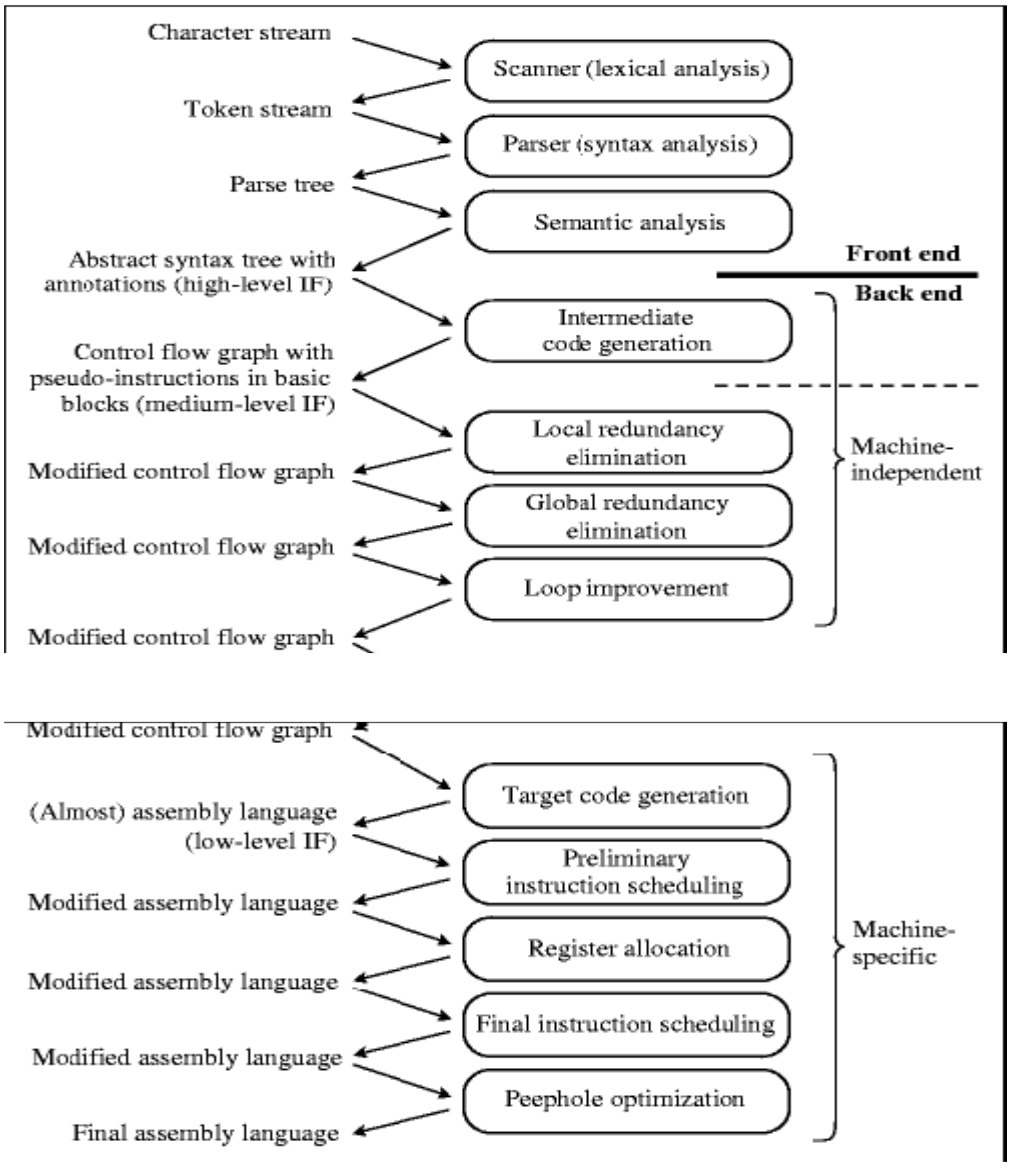
- Optimization is considered "local" if it is done at a basic block level (a sequence of instruction where there is no branch in and out through its entirety).

Global Optimization

- Global Optimization focuses on:
  - Same techniques performed by local optimization but at multi-basic-block level
  - Code modifications to improve the performance of loops.

Both local and global optimization use a control flow graph to represent the program, and a data flow analysis algorithm to trace the flow of information.

Phases of Code Improvement



Peephole Optimization

- Peephole Optimization works by sliding a several-instruction window (a peephole) over the target code, and looking for suboptimal patterns of instructions.
- The patterns to look for are heuristic, and typically based on special instructions available on a given machine.

Elimination of Redundant Loads and Stores

- The peephole optimizer can recognize that the value produced by a load instruction is already available in a register.

Example:

```
r2 := r1 + 5
 i := r2
 r3 := i
 r4 := r3 x 3
```

Elimination of Redundant Loads and Stores

Example:

```
r2 := r1 + 5                    r2 := r1 + 5
 I  := r2          becomes        i := r2
r3 := i                         r4 := r2 x 3
r4 := r3 x 3
```

- Similarly, if there are two stores to the same location within a peephole, then we can eliminate the first.

Constant Folding

• A naïve code generator may produce code that performs calculations at run-time that could actually be performed at compile time.

Example:

```
r2 := 3 x 2          becomes        r2 := 6
```

Constant Propagation

• Sometimes we can tell that a variable will have a constant value at a particular point in a program.

Example:

```
r2 := 4
r3 := r1 + r2
r2 := ...
```

Constant Propagation

• The final assignment to **r2** indicates that its previous value is dead. Loads of dead values can then be cut out.

Example:

```
r2 := 4                         r2 := 4
r3 := r1 + r2      becomes      r3 := r1 + 4
r2 := ...                       r2 := ...
```

then **r3 := r1 + 4**
r2 := ...

Common Sub-expression Elimination

• When the same calculation occurs twice within the peephole, we can often eliminate the second calculation.

Example:

r2 := r1 x 5
r2 := r2 + r3
r3 := r1 x 5

An extra register is often needed to hold the common value.

Example:

r2 := r1 x 5 **r4 := r1 x 5**
r2 := r2 + r3 becomes **r2 := r4 + 3**
r3 := r1 x 5 **r3 := r4**

Copy Propagation

• Even if we cannot tell that the content of register **b** will be constant, we may be able to tell that register **b** will contain the same value as register **a**.

Example:

r2 := r1
r3 := r1 + r2
r2 := 5

• If copy propagation is performed early, it can help decrease register pressure.

Example:

r2 := r1 **r2 := r1**
r3 := r1 + r2 becomes **r3 := r1 + r1**
r2 := 5 r2 := 5

then **r3 := r1 + r1**
r2 := 5

Strength Reduction

• Multiplication or division by powers of two can be replaced by adds or shifts.

Example:

r1 := r2 x 2
r1 := r2 / 2
r1 := r2 x 0

• Numeric identities can sometimes be used to replace a more expensive instruction with a cheaper one.

• Algebraic identities allow us to simplify instructions.

Example:

r1 := r2 x 2 **r1 := r2 + r2**
r1 := r2 / 2 becomes **r1 := r2 >> 1**
r1 := r2 x 0 **r1 := 0**

Elimination of Useless Instruction

• Some instructions that do not modify any memory storage can be dropped.

Example:

r1 := r1 + 0

r1 := r1 x 1

Loop Improvement

• Programs tend to spend most of their time in loops therefore code optimization that can increase the speed of loops are very important.

• These are several techniques to achieve this.

• Removal of loop invariants is one that can be used to extract a part of loop's body that does not change to the loop's header.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Algorithm:-

Execute the program with the following commands.

```
$ gcc codeopt.c
$  ./a.out
```

OR

```
$ lex ic.l
$ yacc ic.y
$ cc y.tab.c -ll
$ ./a.out
```

FAQ's

1. What is code optimization? & why it is required?
2. What are principle sources of code optimization?
3. What is common sub expression elimination?
4. What is compile time evaluation?
5. What is variable propagation?
6. What is dead code elimination?
7. What is loop optimization?
8. What are different methods to carry out loop optimization?
9. What is local & global optimization?
10. What is loop invariant computation?
11. What is copy propagation?

Outcomes Achieved :

Experiment Number: 08

TITLE: Code generation using DAG / labeled tree.0

PROBLEM STATEMENT:

Implement Code generation using DAG / labeled tree.

OBJECTIVE:

1. To understand working of Code Generation Phase of Compiler.

THEORY:

Issues in Code Generation Phase :

Following generic issues are concerned while designing code generator.

(1) Input to code generator
(2) Target program
(3) Memory management
(4) Instruction selection
(5) Register allocation
(6) Choice of evaluation order
(7) Approaches to code generation.

Let us see one by one how these issues are concerned with design of code generation phase of the compiler.

Input to Code Generation Phase (or Input to Code Generator) :

We know that input to code generator is the output of 'Intermediate Code Generator'

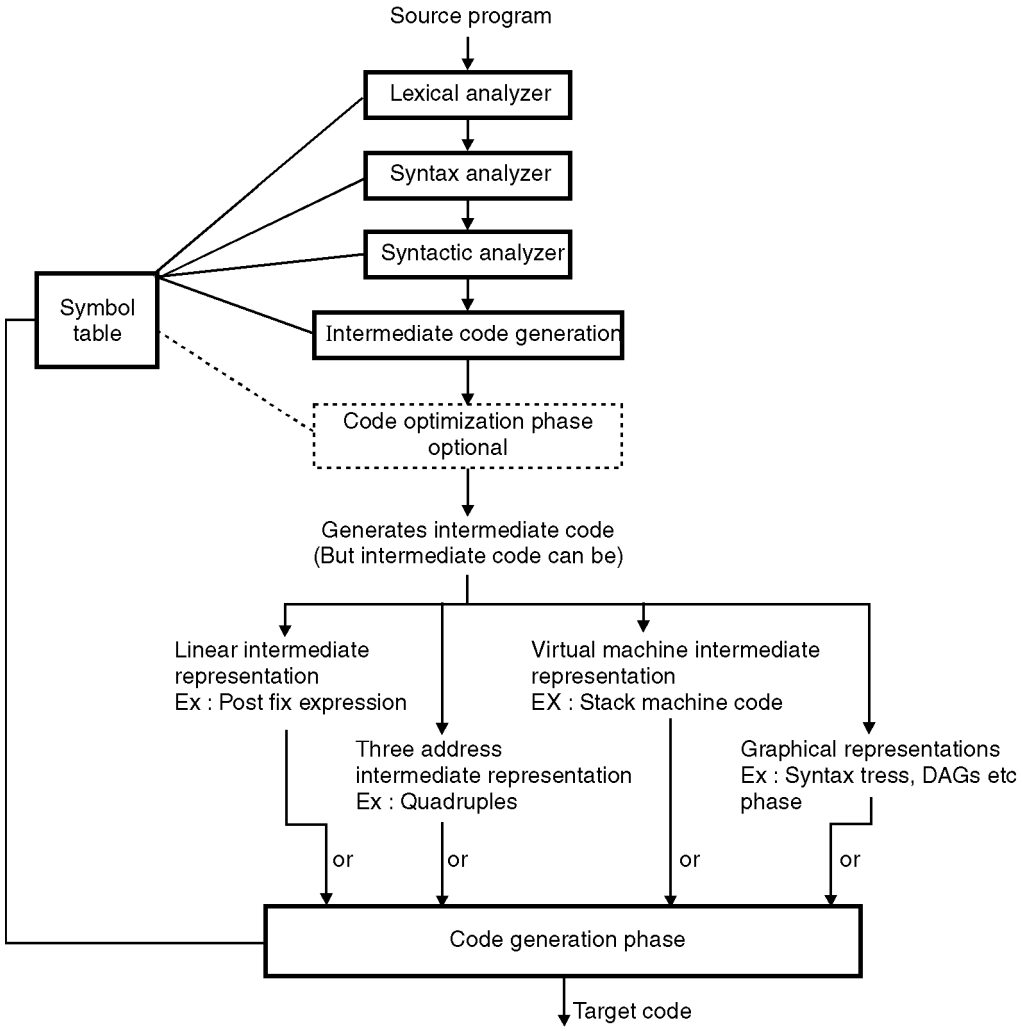
Now output of intermediate code generator phase is:

(a) Intermediate Representation (Intermediate code) :

- This intermediate code is generated by front end of compiler.
- Intermediate code produced by intermediate phase can be any of the following as
  - Qudraples
  - Triples

(b) Information in Symbol Table:

- This information is used to determine runtime address of all data objects which are specified by their names in the input of code generation phase i.e. Intermediate code.



One of possible Intermediate Representations.
Fig.: Input to code generator

- Thus, before code generation phase, source program is processed by Lexical, Syntax, Semantic Analyzers, Intermediate code generator and may or may not be Code Optimizer. Because code optimization can be implemented before and/or after and/or during code generation phase of the compiler.
- Thus, while designing code generator it is assumed that
  (i) Source program is already analysed, parsed and some equivalent intermediate code is generated.
  (ii) Type checking is already performed.

(iii) Intermediate code has some additional type conversion functions.
(iv) All syntactic and semantic errors are detected.
(v) Intermediate code given as input to Code Generation phase is error free.

- As shown in Fig.. intermediate code generated by intermediate code generation phase can be either of a possible types of intermediate representations as
  (a) Linear representation as postfix expressions.
  (b) Three address intermediate representation as quadruples
  (c) Virtual machine representation as stack machine code.
  (d) Graphical intermediate representation as parse tree pr DAGS.
- Now code generator algorithms are different for different intermediate representation of source program i.e. code generator algorithms has to be designed according to intermediate representation generated by intermediate code generator.
- Algorithm discussed can be used for three address code, parse or syntax tees.

Other algorithms can be used for other intermediate representations.

IMPLEMENTATION DETAILS / DESIGN LOGIC:

(Algorithm/Flow Charts/Pseudo Code/DFD/UML diagrams)

Pseudo Code:-

LEX:

Execute the program with the following commands.

```
$ yacc –d Codegen.y
$ lex Codegen.l
$cc lex.yy.c y.tab.c –ll –ly -lm
 $.\a.out
```

FAQ's

1. Design Issues in Code Generation.
2. Types of Target Code s that can be generated by Code Generation Phase.
3. Flow Graph and Basic Block, along with Partition Algorithm.
4. Input to the Code Generation like Instruction Selection, Memory Management, Choice of Evaluation Order etc.
5. Instruction Cycle, Instruction Cost etc.

Experiment Number: 09

TITLE: *Case Study:* Haskell Programming Language